How to use the CAN bus in Linux

# Introduction

The CAN is an ISO standard (ISO 11898) for serial Communications bus originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus. The protocol was developed 1980 by BOSCH. Today CAN has gained widespread use in other areas such as aerospace, industrial automation and [medical](http://en.wikipedia.org/wiki/Medical) equipment [1]

Traditional CAN drivers for Linux are based on the model of character devices. Typically they send and receive raw CAN frames directly to/from the controller hardware through a character device interface. These lead to several drawbacks [2]:

* Changing the CAN hardware vendor often urges the adaption of the CAN application to the new driver’s API.
* Queuing of frames and higher-level protocols have to be implemented in the application.
* Only one application can use a CAN interface at a time.

SocketCAN was designed to overcome these limitations. It uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The communication with the CAN bus is therefore done analogously to the use of the Internet Protocol via sockets. The SocketCAN uses the model of network devices, which allows multiple applications to access one CAN device simultaneously. Also, a single application is able to access multiple CAN networks in parallel [3].

User Space

CAN Application

Linux Socket Layer

PF\_INET

PF\_CAN

CAN\_RAW

...

CAN core

Networking Device Driver

CAN Controller, CAN Transceiver

Kernel Space

Hardware

Figure 1. SocketCAN Architecture

SocketCAN architecture is shown in Figure 1, is mainly divided into three layers, namely, user space, kernel space and hardware.

In order to bring CAN networking to the Linux kernel, two fundamental components of SocketCAN is implemented, the network device drivers for different CAN controllers and a new protocol family PF\_CAN which coexists with other protocol families like PF\_INET for the [Internet Protocol](http://en.wikipedia.org/wiki/Internet_Protocol). The protocol family PF\_CAN provides the structures to enable different protocols on the bus: Raw sockets for direct CAN communication and transport protocols for point-to-point connections [4].

AT91 CAN device driver in Linux is implemented as a networking driver that confirms to the SocketCAN interface.

This application note explains how to enable the CAN controller in Linux using as an example on SAMA5D34-EK[7] and how to use it. And we also provide some application code shows how to open a socket and send and receive data to/from CAN bus using SocketCAN.

# How to Configure

## Kernel Layer Configuration

The CAN device driver is currently build-into the kernel with the right configuration items enabled. We take Linux-3.6.9 for reference.

It is available as the configuration items in the Linux kernel configured as follows.

[\*] Networking support -🡪

<\*> CAN bus subsystem support -🡪

<\*> Raw CAN Protocol (raw access with CAN-ID filtering)

<\*> Broadcast Manager CAN Protocol (with content filtering)

CAN Device Drivers -🡪

<\*> Virtual Local CAN Interface (vcan)

<\*> Platform CAN drivers with Netlink support

[\*] CAN bit-timing calculation

<\*> Atmel AT91 onchip CAN controller

If the configuration item, “CAN bit-timing calculation” is not configured, the CAN bit-timing parameters need to be set individually. When this item is configured, the CAN bit-timing parameters will be calculated if the bit-rate is specified with the argument “bitrate”. So, we recommended that this item should be configured.

The AT91 CAN driver can also be configured as a loadable module.

### Device Tree Node

After CAN driver is enabled in the kernel, the controllers (CAN0 and CAN1) still need to be configured in the device tree file.

The CAN device tree configuration includes the CAN node with its properties and pinctrl property configuration. About these property meaning please refer to Linux Documentation [5][6].

Snapshot of *sama5d3.dtsi.*

can0 {

pinctrl\_can0\_rx\_tx: can0\_rx\_tx {

atmel,pins =

<3 14 0x3 0x0

3 15 0x3 0x0>;

};

};

can1 {

pinctrl\_can1\_rx\_tx: can1\_rx\_tx {

atmel,pins =

<1 14 0x2 0x0

1 15 0x2 0x0>;

};

};

can0: can@f000c000 {

compatible = "atmel,at91sam9x5-can";

reg = <0xf000c000 0x300>;

interrupts = <40 4 3>;

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_can0\_rx\_tx>;

status = "disabled";

};

can1: can@f8010000 {

compatible = "atmel,at91sam9x5-can";

reg = <0xf8010000 0x300>;

interrupts = <41 4 3>;

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_can1\_rx\_tx>;

status = "disabled";

};

Generally, the basic idea is that what's inside the SoC is identical across all boards using that SoC, This information is put into the .dtsi file so it can be included by the .dts file for any boards using the SoC. Normally, you could use it without modification.

Your board-specific setting goes into the board's individual .dts file. For the properties existing in both, the most recent value in parsing order overrides any earlier values.

Go back to here, the SoC level *sama5d3.dtsi* file said that 'status = "disabled"' for CAN node, the board-level *sama5d34ek.dts* could later override it by saying 'status = "okay"' within the same node.

Snapship of *sama5d34ek.dts*

can0: can@f000c000 {

status = "okay";

};

can1: can@f8010000 {

status = "okay";

};

## User space configuration and utilities

Since CAN is a networking interface and uses the socket layer concepts, many utilities have been developed in open source for utilizing CAN interfaces. These utilities are also very useful for debugging the driver. We focused on the following two utility sets;

* iproute2: a set of utilities is to configure the CAN interface netlink standard utilities
* can-utils: CAN user space utilities and tools for [SocketCAN](http://server/twiki/bin/edit/Shanghai/SocketCAN?topicparent=Shanghai.LinuxCAN;nowysiwyg=0" \o "SocketCAN (this topic does not yet exist; you can create it)).

Iproute2 is a collection of utilities for controlling [TCP](http://en.wikipedia.org/wiki/Transmission_Control_Protocol) and [UDP](http://en.wikipedia.org/wiki/User_Datagram_Protocol) [IP](http://en.wikipedia.org/wiki/Internet_Protocol) networking and [traffic control](http://en.wikipedia.org/wiki/Network_traffic_control) in [Linux](http://en.wikipedia.org/wiki/Linux), in both [IPv4](http://en.wikipedia.org/wiki/IPv4) and [IPv6](http://en.wikipedia.org/wiki/IPv6) networks. It is intended to replace an entire suite of standard Unix networking tools (often called “net-tools”) that were previously used for the tasks of configuring network interfaces, [routing](http://en.wikipedia.org/wiki/Routing) tables, and managing the [ARP](http://en.wikipedia.org/wiki/Address_Resolution_Protocol) table[8].

The can-utils includes cansend, cangen or candump tools to send and receive packets via CAN interface to test CAN.

The root file system image is produced via using buildroot-2013.02 for our project. Fortunately, the iproute2 and can-utils utilities packages have been supported. They are compiled and installed on the target with the right configuration items enabled.

In fact, in the buildroot, they could be configured as follows.

Package Selection for the target -🡪

Networking applications -🡪

[\*] can-utils

[\*] iproute2

After successfully building, the root file system contains /sbin/ip and the other CAN tools like cansend, cangen or candump.

These tools can be verified using command.

# which ip

/bin/ip

# which cansend

/usr/bin/cansend

# which candump

/usr/bin/candump

# Use SocketCAN with existing commands

## Hardware Setup

There are two CAN controllers for SAMA5D3. We will use these two CANs to illustrate how to use SocketCAN. At first, the following connections needs be made to connect these two CAN interfaces as shown in figure 2.

CANTX1

CANRX1

CANRX0

CANTX0

CAN0

CAN1

CAN

Transceiver0

CAN

Transceiver1

CANH

CANL

SAMA5D3X

Figure 2. Hardware Connection

## Configure CAN interface

* Verify CAN support

After the kernel and buildroot configuration are done, then build, flash the board, power up.

To Verify CAN support configuration, using the command to look for the following message in the Linux boot output.

# dmesg | grep can

vcan: Virtual CAN interface driver

at91\_can f000c000.can: device registered (reg\_base=e0d28000, irq=20)

at91\_can f8010000.can: device registered (reg\_base=e0d2a000, irq=28)

qt1070 1-001b: can not read register, returned -121

can: controller area network core (rev 20120528 abi 9)

can: raw protocol (rev 20120528)

can: broadcast manager protocol (rev 20120528 t)

If the AT91 CAN driver was configured as a loadable module, you can install it. Its name is at91\_can.ko.

# insmod at91\_can.ko

at91\_can f000c000.can: device registered (reg\_base=e0dce000, irq=20)

at91\_can f8010000.can: device registered (reg\_base=e0dd0000, irq=28)

* Examine the CAN network interface using command.

# /sbin/ip -details link show | grep can

2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT qlen 10

link/can

can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0

at91\_can: tseg1 4..16 tseg2 2..8 sjw 1..4 brp 2..128 brp-inc 1

3: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT qlen 10

link/can

can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0

at91\_can: tseg1 4..16 tseg2 2..8 sjw 1..4 brp 2..128 brp-inc 1

There are two CAN network interfaces available, can0 and can1, showed as the above message.

* Set the bitrate

Set the can0 and can1 bit rate to 125Kbits/sec

# /sbin/ip link set can0 type can bitrate 125000

# /sbin/ip link set can1 type can bitrate 125000

Once the driver installed and the bitrate is set, the CAN interface can be started like a standard net interface.

* Start the can0 and can1 using the command:

# /sbin/ip link set can0 up

at91\_can f000c000.can: can0: writing AT91\_BR: 0x00200561

# /sbin/ip link set can1 up

at91\_can f8010000.can: can1: writing AT91\_BR: 0x00200561

## Use CAN interface

* Can-utils usage

To test the CAN interface, transfer and receive simple packets by using cansend and candump tools from can-utils utility.

candump: dump traffic on a CAN network.

cansend: simple command line tool to send CAN-frames via CAN\_RAW sockets.

The cansend usage:

# cansend -h

Usage: cansend <device> <can\_frame>.

Here the device is the CAN network interface name, typically can0 or can1, and a CAN frame is in the format:

<can\_id>#{R|data}

Here, the can\_id having 3 (SFF) or 8 (EFF) hex chars. and data in the format of zero to eight 8-bit hex-values that can optionally be separated by a period ('.') or use R for remote transmission request.

The candump usage:

# candump -h

Usage: candump [options] <CAN interface>+

(use CTRL-C to terminate candump)

* Test CAN using can-utils tools

To receive the CAN data frame package, run candump on the can1 interface.

To Transfer a CAN data frame message, with a can\_id arbitration field value of 0x123 and a data field value 0x11,0x22,0x33, 0x44, 0x55, 0x66, 0x77, 0x88, run cansend on the can0 interface.

# candump can1 &

# cansend can0 123#11.22.33.44.55.66.77.88

can1 123 [8] 11 22 33 44 55 66 77 88

* About procf for SocketCAN

Many information about SocketCAN can be retrieved through procfs, such as, socket receive filter lists, SocketCAN core statistics, the SocketCAN core version.

The procf content for SocketCAN can be listed using command

# ls /proc/net/can/

rcvlist\_all rcvlist\_err rcvlist\_inv reset\_stats version

rcvlist\_eff rcvlist\_fil rcvlist\_sff stats

The SocketCAN statistics can be retrieved using command.

# cat /proc/net/can/stats

1 transmitted frames (TXF)

2 received frames (RXF)

1 matched frames (RXMF)

50 % total match ratio (RXMR)

0 frames/s total tx rate (TXR)

0 frames/s total rx rate (RXR)

50 % current match ratio (CRXMR)

0 frames/s current tx rate (CTXR)

0 frames/s current rx rate (CRXR)

50 % max match ratio (MRXMR)

1 frames/s max tx rate (MTXR)

2 frames/s max rx rate (MRXR)

2 current receive list entries (CRCV)

3 maximum receive list entries (MRCV)

# Using SocketCAN with Own Application

The simplest of methods to access the CAN bus is to send and receive raw CAN frames. This programming interface, which is similar to the character device drivers, is offered by the CAN\_RAW protocol.

First Create a socket for communicating over a CAN network with the socket(2) system. Then bind the socket to a CAN interface using bind(2) system call. Optionally the application specific filter lists should be used to control the socket with the CAN\_RAW\_FILTER socket options.

Finally uses the standard system calls to read and write CAN frames, represented by the “struct can\_frame” defined in include/Linux/can.h.

The application example code is shown as following.

*cansend\_test.c*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <net/if.h>

#include <sys/ioctl.h>

#include <sys/socket.h>

#include <linux/can.h>

#include <linux/can/raw.h>

int main(int argc, char \*\*argv)

{

int sock;

struct sockaddr\_can addr;

struct can\_frame frame;

struct ifreq ifr;

int ret = 0;

if (argc != 2) {

printf("Usage: %s <device>.\n", argv[0]);

return -1;

}

if ((sock = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW)) < 0) {

perror("socket");

return 1;

}

addr.can\_family = AF\_CAN;

strcpy(ifr.ifr\_name, argv[1]);

if (ioctl(sock, SIOCGIFINDEX, &ifr) < 0) {

perror("SIOCGIFINDEX");

ret = 1;

goto error;

}

addr.can\_ifindex = ifr.ifr\_ifindex;

if (bind(sock, (struct sockaddr \*)&addr, sizeof(addr)) < 0) {

perror("bind");

ret = 1;

goto error;

}

frame.can\_id = 0x1;

frame.can\_dlc = 8;

frame.data[0] = 0xa1;

frame.data[1] = 0xa2;

frame.data[2] = 0xa3;

frame.data[3] = 0xa4;

frame.data[4] = 0xa5;

frame.data[5] = 0xa6;

frame.data[6] = 0xa7;

frame.data[7] = 0xa8;

printf("CAN send data on: %s.\n", ifr.ifr\_name);

if (write(sock, &frame, sizeof(frame)) != sizeof(frame)) {

perror("write");

ret = 1;

goto error;

}

error:

close(sock);

return ret;

}

*canread\_test.c*

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <signal.h>

#include <net/if.h>

#include <sys/socket.h>

#include <sys/ioctl.h>

#include <linux/can.h>

#include <linux/can/raw.h>

int main(int argc, char \*\*argv)

{

int sock;

struct sockaddr\_can addr;

struct can\_frame frame;

struct ifreq ifr;

int nbytes;

unsigned char loop = 1;

struct can\_filter filter[2];

unsigned char i, length;

unsigned int index = 0;

int ret = 0;

if (argc != 2) {

printf("Usage: %s <device>.\n", argv[0]);

return -1;

}

sock = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);

if (sock < 0) {

perror("socket");

return 1;

}

addr.can\_family = AF\_CAN;

memset(&ifr.ifr\_name, 0, sizeof(ifr.ifr\_name));

strcpy(ifr.ifr\_name, argv[1]);

if (ioctl(sock, SIOCGIFINDEX, &ifr) < 0) {

perror("SIOCGIFINDEX");

ret = 1;

goto error;

}

addr.can\_ifindex = ifr.ifr\_ifindex;

addr.can\_ifindex = if\_nametoindex(ifr.ifr\_name);

if (bind(sock, (struct sockaddr \*)&addr, sizeof(addr)) < 0) {

perror("bind");

ret = 1;

goto error;

}

filter[0].can\_id = 0x1;

filter[0].can\_mask = CAN\_SFF\_MASK;

setsockopt(sock, SOL\_CAN\_RAW, CAN\_RAW\_FILTER,

&filter, sizeof(filter));

while (loop) {

nbytes = read(sock, &frame, sizeof(struct can\_frame));

if (nbytes < 0) {

perror("read");

ret = 1;

loop = 0;

}

length = frame.can\_dlc;

printf("CAN read frame on: %s\n", ifr.ifr\_name);

printf("No.%d: FrmID = 0x%x DLC = %d, data:",

index++, frame.can\_id, frame.can\_dlc);

for (i = 0; i < length; i++) {

printf(" 0x%x", frame.data[i]);

}

printf("\n");

}

error:

close(sock);

return ret;

}

Run canread\_test on the CAN1 interface to read data, and run cansend\_test on the CAN0 interface to send the CAN frame. The result is shown.

# ./canread\_test can1 &

# ./cansend\_test can0

CAN send data on: can0.

CAN read frame on: can1

No.0: FrmID = 0x1 DLC = 8, data: 0xa1 0xa2 0xa3 0xa4 0xa5 0xa6 0xa7 0xa8

# Inside AT91 CAN Networking Device Driver

The driver code is available at drivers/net/can/at91\_can.c of kernel code tree. Similar to other known network device drivers, there are three parts which deal with:

1. The CAN networking device driver initializes and configures the hardware, finally with “register\_candev()” the CAN device is registered at the networking subsystem.
2. With the reception of a CAN frame the controller issues an interrupt and Linux will execute the registered interrupt handler, or schedule a routine to read multiple frames later in a software IRQ context (called “NAPI”), and “netif\_receive\_skb()” or “netif\_rx()” function is called to push the CAN frame into the packet scheduler.
3. The transmission of a CAN frame is originated in the local system, usually in the user space. When the CAN frame is passed into the packet scheduler, “start\_xmit()” function will be called to activate the transmission of the CAN frame.

Usually, the following two points are most interesting:

1. CAN bit-timing parameters setting.

The CAN networking layer core contains an algorithm to calculate the actual bit timing parameter based on the requested bit rate, the current CAN clock and the bit timing parameters.

So the structure is “struct bittiming\_const” should be filled, this structure describes the bit timing limits (tseg1, tseg2, …) of the hardware in multiple of the Time Quantum, i.e. in a clock rate independent way. You may have to adjust these timings according to the specific clock of your boards.

1. AT91 CAN hardware filter.

AT91 CAN controller supports hardware filter for each mail box, but the filters on driver level would affect all users in the multi-user system. SocketCAN does not support hardware filtering of incoming CAN frames. All CAN frames are received and passed to the CAN networking layer core, which processes the application specific filter lists. The hardware filter should be disabled.

Using hardware filter could be helpful for system’s efficiency. If you want to use a limited number of mail box by hardware filter, you may adopt these part of code according to your need.

# Conclusion

As said above, the SocketCAN presents the developer a multi application capable, standard socket based API to send and receive raw CAN frames independent from the used CAN controller. It further offers the driver developer a standard network driver model known from Ethernet drivers.

# References

[1] Wiki CAN bus [Online]. Available: <http://en.wikipedia.org/wiki/CAN_bus>

[2] Marc Kleine-Budde, Pengutronix, “SocketCAN - The official CAN API of the Linux kernel,”

[Online] <http://www.can-cia.org/fileadmin/cia/files/icc/13/kleine-budde.pdf>

[3] Wiki SocketCAN. [Online]. Available: <http://en.wikipedia.org/wiki/SocketCAN>

[4] Linux Documentation. Documentation/networking/can.txt

[5] Linux Documentation. Documentation/devicetree/bindings/pinctrl/atmel,at91-pinctrl.txt

[6] Linux Documentation.Documentation/devicetree/bindings/net/can/atmel-can.txt

[7] Atmel website:<http://www.atmel.com/products/microcontrollers/arm/sama5d3.aspx>

[8] Wiki iproute2. [Online]. <http://en.wikipedia.org/wiki/Iproute2>

# Revision History

|  |  |  |
| --- | --- | --- |
| Doc. Rev. | Comments | Change Request Ref. |
|  | First issue |  |